Synchronization primitives

Priit Järv, Tallinn University of Technology

2014

1 Introduction

These notes are to accompany lectures on parallel programming and synchronization. They should be suitable for individual study, although some understanding of programming and computer hardware is expected of the reader. In Section 2 we walk through a simple example of a situation where synchronization is useful. Section 3 gives an overview of synchronization methods offered by operating systems and some programming languages.

If the reader is already familiar with the above, then they may wish to skip to Sections 4–7 that deal with the subject matter. We describe implementation details of spinlocks, an algorithm for a reader-writer lock and a lock-free solution to the queue problem presented in Section 2.

2 A brief introduction to synchronization

Most modern computers (in this context, this includes cell phones and the likes) we come across are multiprocessing systems. Many even have multiple processors, although this is not a requirement. Even so, a lot of programs quietly mind their own business, executing in a single process and not causing any trouble. Then there are programs that are trying to do many things in parallel, either because they need to or because the programmer thought they need to. Experienced programmers like to tell newbies that this is a dangerous and complicated affair best left to people who know what they are doing. Of course, this will only encourage the beginning programmer more.

To do things in parallel, a program must have many processes or threads. They are different things, but in the context of this article the differences are not important. Synchronization comes in whenever several threads or processes have a common resource they want to access - a file, a memory area or maybe a peripheral device.

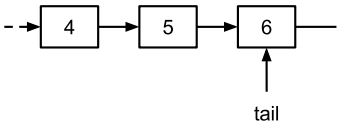
As is customary, we will begin with a cautionary tale. This tale is about race condition, the nemesis of all programmers - except perhaps those looking to poke holes in others' programs.

Let's take a FIFO queue. It is a data structure different from stack in that if you put something in, it comes out from the other end instead. So there is

something called the head of the queue, where things come out and the tail, where things go in. When implementing the queue as a linked list, head and tail are probably going to be pointers.

This time we will forget about the head of the queue and only look at the tail (Figure 1). Adding elements to the queue means appending them to the linked list. Of course we will have two threads that add to the queue. This is already enough to get a big ugly race condition.

Figure 1: Tail of the FIFO



Our declaration of the linked list will be minimalistic (Listing 1).

Listing 1: FIFO element

```
struct elem {
  int data;
  struct elem *next;
};
```

The function to add elements is quite simple too (Listing 2). The `next` pointer of the last element is set to point to the new element. Then we also point the `tail` pointer to the new element, so that we can find the correct place for the next addition.

Listing 2: Adding elements to the FIFO

```
void add_elem(struct elem **tail, struct elem *e) {
  (*tail)->next = e;
  *tail = e;
}
```

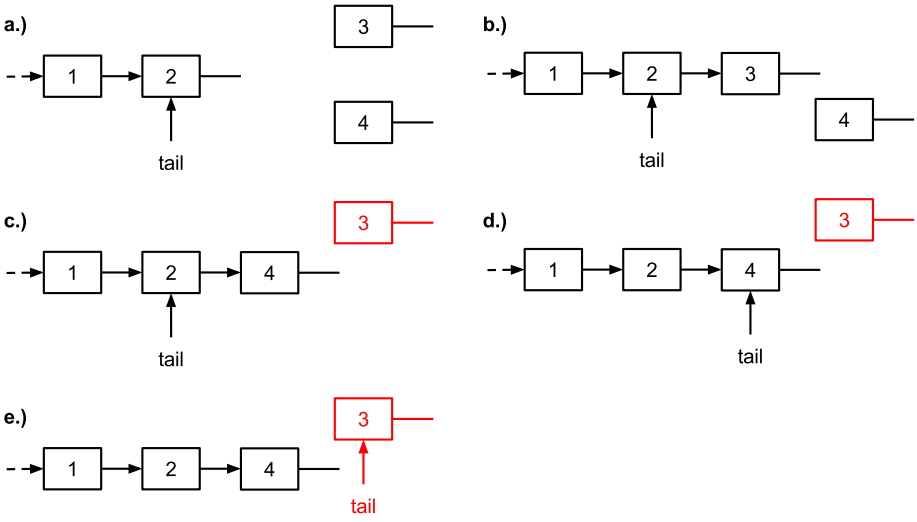
There are just two lines of program code, but there is plenty of space for *multiple* race conditions to occur. The disassembly of those two lines (Figure 2) shows everything in plain sight. The addresses of the tail pointer and the new element come from the stack which is fine - we know neither will change. What interests us here is the one read from an outside memory address, followed by two writes (in red).

Figure 2: Disassembly of add_elem()

```
1  mov     -0x8(%rbp),%rax     /* RAX <- tail pointer address (stack) */
2  mov     (%rax),%rax         /* RAX <- tail elem address */
3  mov     -0x10(%rbp),%rdx    /* RDX <- address of new elem (stack) */
4  mov     %rdx,0x8(%rax)      /* tail elem+8 bytes offset <- RDX */
5
6  mov     -0x8(%rbp),%rax     /* RAX <- tail pointer address (stack) */
7  mov     -0x10(%rbp),%rdx    /* RDX <- address of new elem (stack) */
8  mov     %rdx,(%rax)         /* tail pointer <- address of new elem */
```

The first race condition and the root of all trouble is on line 2 of the listing. If one thread reads the `tail` pointer and doesn't yet reach line 8 before another thread executes line 2, two threads have the same value of the pointer. Then a new race to line 4 begins - who can overwrite the `next` pointer of the current last element. Whoever is first, loses! Their element will be orphaned. The final race is to overwrite the `tail` pointer. Again the one to arrive later gets the last word. The damage ranges from losing one element to the whole queue getting chopped in two (Figure 3).

Figure 3: FIFO corruption. a.) New elements "3" and "4" created. b.) *t1* adds "3" to the tail element. c.) *t2* adds "4" to the tail element; "3" is orphaned. d.) *t2* updates tail pointer. e.) *t1* updates tail pointer; queue is split.



The instructions on lines 2-8 are commonly called a "critical section". Evidently, if more than one thread is executing that part at the same time, bad things happen. If we want to avoid that, the threads should be *synchronized*

to one another so that if one is in the critical section, others wait. This is such a common need that computer hardware, operating systems and several programming languages all contain their own methods of synchronization.

3 Available methods

Semaphores have been around for ages, which is why they are readily available on UNIX-like OS-es. They come in two flavours, the older System V which has become standard through contemporary OS-es seeking to support it; and POSIX IPC which is actually a standard.

A semaphore is a counter. If it is positive, the resource protected by it is available, when it falls to zero, access to the resource is blocked. Of course, the semaphore does not physically block access to anything. It merely cannot be decremented below 0.

As both IPC (Inter-Process Communication) API-s are similar, we'll describe the POSIX API (Listing 3). Here we use a semaphore in local memory, which is enough if we want to synchronize between threads. For processes, shared memory is the only way. Each time a thread enters the critical section, it will call `sem_wait()`. If the semaphore can currently be decremented, the call will return, otherwise it will block until some other thread calls `sem_post()`.

Listing 3: POSIX semaphore

```
/* initialize */
sem_t s;
sem_init(&s, 0, 1);                    /* local semaphore, set semaphore=1 */

/* use */
if (!sem_wait(&s)) {                              /* --semaphore; "lock" */
  /* ... critical section ... */
  sem_post(&s);                                  /* ++semaphore; "unlock" */
}
```

Note that the initial value can be greater than 1, but for protecting the critical section we obviously need to choose 1.

The POSIX threads API, commonly called `pthreads`, provides it's own, rich set of synchronization methods [1]. The main workhorse is the mutex, which is used the same way as we used the semaphore (Listing 4). Beginning to see the pattern?

By the way, with `pthreads` we are not limited to synchronizing inside a single process. True, this is the default behaviour, but a mutex or other synchronization variable may be configured to reside in shared memory and may live beyond the lifetime of the process creating it.

[1] `https://computing.llnl.gov/tutorials/pthreads/`

Listing 4: Mutex with pthread library

```
/* initialize */
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

/* use */
if (!pthread_mutex_lock(&m)) {
  /* ... critical section ... */
  pthread_mutex_unlock(&s);
}
```

In Windows native C++ API, the semantics are again similar (Listing 5). Microsoft has piled on functionality during the years and currently the feature set rivals that of `pthreads` [2].

Listing 5: Mutex in Windows native API

```
/* initialize */
HANDLE m = CreateMutex(NULL, FALSE, NULL);

/* use */
if (WaitForSingleObject(m, INFINITY) == WAIT_OBJECT_O) {
  /* ... critical section ... */
  ReleaseMutex(m);
}
```

This mutex pattern is probably getting repetitive, so we'll next turn to programming languages, where there is a great deal of variety. Ada (very unfashionable) and Go (very fashionable) are good examples of builtin parallel programming support. With limited space available we will look at Java instead.

Thread synchronization and thread support in general is rather elegantly solved in Java. We can declare that we are doing something that should be synchronized between threads (Listing 6).

Admittedly, the given example is a bit artificial, because Java comes out of the box with a selection of classes that implement standard data structures, several of which are already thread-safe[3].

4 The infamous spinlock

Do I need to program my own primitives? Often, the answer is "no". However, it is enlightening to learn how to. What better place to start with than

[2] `http://msdn.microsoft.com/en-us/library/windows/desktop/ms686353%28v=vs.85%29.aspx`

[3] `http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html`

Listing 6: synchronized keyword in Java

```
class FIFO {
  LinkedList<Object> queue;
  FIFO() {
    this.queue = new LinkedList<Object>();
  }
  synchronized void add(Object e) {
    queue.add(e);
  }
}
```

the spinlock. In its simplest form it takes a couple of assembly instructions (Listing 7).

Listing 7: Spinlock with test-and-set

```
syn_var:
        .quad 0x0
spin_wait:
        lock btsq $0, syn_var        /* read and set LSB of syn_var atomically */
        jc spin_wait                      /* jump back if LSB was already set */
        /* critical section */
        movq $0, syn_var              /* clear LSB (by writing a 64-bit zero) */
```

The `syn_var` is just a 64-bit area of memory. If it contains 1 (i.e. the LSB is set), some process is in the critical section; if it is 0 then a new process may enter. Note the `BTS` atomic instruction (the "q" suffix is just GNU assembler syntax saying that the operand is 64-bit). Basically it is a way to write to memory, just like `MOV` does, but with the added perk that we can check later what the overwritten memory contents were[4].

So, the trick is to start by rushing in and setting the state to "locked". Overwriting 1 with 1 does not change anything, so this is acceptable even if some other process already holds the lock. Then we have time to check in peace and quiet whether we actually did get the lock. If the previous value was 0, we were successful and everybody else is now waiting after us. Otherwise, we have to keep trying.

This kind of repetitive re-checking of the synchronization variable is what the "spinning" in the name refers to. It is also called the "busy wait". The idea that the processor is stuck in a loop like this, doing no useful work, does not sit well with many people. In case of single-core computers the spin wait truly is useless - until there is a context switch, i.e. another process or thread gets the CPU, there is no hope that the lock is released anyway.

Fortunately, the simple version we started with can be improved. The first

[4]The Intel instruction set has the quirk that this instruction is also bitwise.

optimization is yielding the CPU to other processes if it looks like getting the lock is going to take long. To do so, a spin count is added. After the spin count limit is reached, the lock routine calls `usleep()` (or something similar), which suspends the execution of current thread. If the spin count is in the low hundreds, the performance loss from spinning should be insignificant compared to the cost of context switches. Of course, there are still situations where the lock holder is simultaneously active on another CPU. In that case we'd like to have a reasonable time window to give it a chance to exit the critical section and release the lock while we're still spinning.

5 Spin-waits on modern hardware

Many things that hardware manufacturers have put in the processors to make them go fast make our spin-wait loop slower, paradoxically. The blame lays mostly with the how the cache memory is used. Synchronization variables, by definition, are accessed by multiple threads or processes, which may run on multiple cores. This means that the variable is also present in the caches of multiple cores.

If a core writes to a memory location, the cache coherency protocol kicks in. All copies of the value in the caches of other cores must be marked invalid. Not only does this consume system resources, but when it is the turn of these cores to access the variable, they no longer have it ready and must go asking around for it on the CPU interconnect network[2]. If they intend to write to the variable then this will additionally cause the "flushing" of the latest value into the main memory. But the most likely scenario where this happens is when multiple threads are spin-waiting, meaning that the writes are being spammed relentlessly by multiple cores.

Most of this unpleasantness can be avoided by checking if the variable is in the unlocked state before attempting the atomic test-and-set (Listing 8).

Listing 8: Avoiding cache line invalidation at each iteration

```
spin_wait:
        cmpq $0, syn_var                    /* syn_var == 0? */
        jne spin_wait                       /* if not, go read again */
        lock btsq $0, syn_var
        jc spin_wait
        /* critical section */
        movq $0, syn_var
```

This replaces large majority of writes, causing all the interconnect network (or system bus on older systems) traffic, with reads from local cache. The readers with sharp eyes might have noticed a race condition: between `CMP` and `BTS` instructions `syn_var` can change. This is inconsequential, because the whole point of the spinlock is to race to grab the lock. We want *some* thread to get the lock, not a *specific* thread.

We are making progress, but more work needs to be done. Intel has published a memo with recommendations specifically for spin-wait loops [3] on Pentium 4 and newer processors. They describe two new ways the performance can suffer:

- out-of-order execution. A spin-wait loop, like the one above, continuously reads the memory where the synchronization variable is stored. These reads may be executed in arbitrary order, which is fine as we're reading the unmodified value in the cache anyway. However, once another thread updates the value, the processor must sort out whether the reads and the write were done in correct order. This is slow [3].
- False sharing. Caches operate with a minimum unit size that is transferred at a time, called a "cache line". There is no common size but the typical sizes are 64 or 128 bytes - much larger than the synchronization variable. A tidy programmer might place all their synchronization variables in one place, which greatly increases the probability that multiple locks share a single cache line. Then, a write to one variable also causes other cached variables to become invalid, making other cores to re-read them, which in turn forces the freshly written variable to be flushed to main memory - all of this unnecessary because the variables were unrelated! The same scenario can occur if the synchronization variable happens to be on the same cache line with the data that is protected by it.

The first issue is worked around by the `PAUSE` instruction (Listing 9), designed specifically for optimizing spin loops. The processor has some flexibility in what the instruction actually does, but the initial implementation was to cause a delay long enough so that the reads are issued at approximately the same rate the main memory operates - it is pointless to make multiple reads in the span of time of a single write [3]. The loop can be written so that it skips the `PAUSE` during the first iteration.

Listing 9: Using the PAUSE instruction

```
        cmpq $0, syn_var                    /* check for the uncontended case */
        je try_lock                                        /* skip the pause */
spin_wait:
        pause                                              /* spin loop hint */
        cmpq $0, syn_var
        jne spin_wait
try_lock:
        lock btsq $0, syn_var
        jc spin_wait
        /* critical section */
        movq $0, syn_var
```

Sharing cache lines is avoided by placing the synchronization variables so that they each end up on a different cache line. This can be done by aligning each variable to the boundary of a cache-line sized chunk of memory. Since the

addresses start at 0, assuming that the cache line size does not exceed 128, the suitable addresses are multiples of 128 (Listing 10) [3]. Keep in mind though that the cache size is limited. Padding everything like this may consume so much space that the program actually becomes slower.

Listing 10: Alignment to cache line boundary

```
struct syn_str { syn_var_t syn_var; };
void *p = malloc(sizeof(struct syn_str) + 127);
syn_str * align_p = (syn_str *)(((( ptrdiff_t ) p) + 127) & (ptrdiff_t) -128);
/* align_p->syn_var is now aligned properly */
```

The value -128 in two's complement format is 0xff...ff80 regardless of the size of the pointer, in other words a binary number of all 1-s except the last 7 bits. Logical AND between the address and this number gives a multiple of 128. The trouble is though, that by setting lower bits to 0, the value decreases, at most by 127. So if we increment it by 127 in advance, it is guaranteed to remain in the area we have allocated. This alignment trick is useful in other applications as well.

To summarize, we'll present a spinlock implementation that implements most of the above techniques (Listing 11). The macro `MM_PAUSE` will need to be defined separately to emit the `PAUSE` instruction in a platform-dependent way. Also, `compare_and_swap` is a wrapper to the compiler intrinsic implementing the atomic operation with the same name, or CAS in short. It differs from test-and-set in that it only overwrites the memory if it contains an expected value, in this case 0. It is a more powerful atomic operation that we will continue to use from this point.

6 Spinning reader-preference lock

Sometimes the resource we're protecting is not a critical section, but data. Using a mutex (like the spinlocks we described) will give correct results, but data has a special property - it only needs to be protected if we modify it. Reading in parallel is harmless.

The reader-writer lock is a primitive with this specific purpose in mind. It provides two services: the read lock and the write lock. A read lock can be acquired if there are no lock holders or there are only other read locks. A write lock can be acquired if there are no other locks of any type held.

The reader-writer lock may come in several flavours. A reader preference lock gives priority to the readers. It already follows from the read/write lock semantics we've given that if a reader is present, all new read requests will continue to be served, while writers will only get the chance to acquire the lock once the last active reader has finished; this is how the reader preference lock was originally formulated [1][5]. A writer preference lock has an additional rule:

[5] The method of Courtois et al. uses semaphores but leaves their exact semantics undefined.

Listing 11: Complete spinlock implementation in C

```
int i;
struct timespec ts;
volatile syn_var_t *syn_var;

/* Set syn_var to point to the synchronization variable (not shown) */

/* First attempt at getting the lock without spinning */
if (compare_and_swap(syn_var, 0, 1))
      return;

ts.tv_sec = 0;
ts.tv_nsec = SLEEP_NSEC;

/* Spin loop */
for (;;) {
  for (i=0; i<SPIN_COUNT; i++) {
    MM_PAUSE
    if (!(*syn_var) && compare_and_swap(syn_var, 0, 1))
      return;
  }

  /* Backoff */
  nanosleep(&ts, NULL);
  ts.tv_nsec += SLEEP_NSEC;
}
```

if there is a writer waiting, no new read locks will be granted. This has the opposite effect. Finally, the fair lock grants requests on first-come first-serve basis, but consecutive readers can still work in parallel.

We will implement the reader-preference lock, as it is the simplest. The algorithm we'll use was published as a point of comparison for more sophisticated methods [4], but is usable in practice.

Before copy-pasting program code blindly, let's try to reason how this lock could be constructed. At first look, we seem to have three states - the lock is uncontended, there is one or more readers or there is one writer active. We also understand that if we are to use spin-waits, we must go from one state to the other atomically, meaning that the reading of the state and setting a new state must occur with one atomic operation. So, all those states should to be encoded into a single synchronization variable [6].

So, let's propose the values 0, 1 and 2 for "available", "writer active" and "readers active" state, respectively. Creating the write lock procedure now looks straightforward - if the state is 0, set it to 1 and continue. Otherwise spin-wait.

So it is also undefined what happens when the lock is released with both readers and writers waiting, in an actual program the result would depend on the semaphore implementation.

[6] Atomic operations on multiple variables are not generally available.

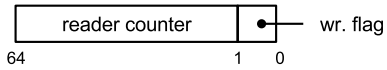
What about the read lock procedure? It seems that 0 and 2 are both good to continue. We can check for both states together in one operation by checking the least significant bit (LSB) of the synchronization variable - it is always 0 for even numbers in binary form; the undesirable state 1 is an odd number. If we were successful then the state will now be 2. When the reader finishes, we check if there are other readers and only set it back to 0 if there were none.

How to check if there are other readers? Here the proposed solution fails spectacularly, because there is no obvious way. The check and the state update must also be atomic, so some sort of reader state table or other outside source is out of the question. The information about the readers in the synchronization variable is completely anonymous and unhelpful, too.

It turns out that we can live with the anonymity of the readers and the synchronization variable can still be quite helpful in keeping track of them. The solution is to make a counter of readers. Each reader is responsible for incrementing the counter when they start reading and decrementing it when they're done. This way it doesn't matter in which order the readers come and go, when they've all finished the counter is at 0. As long as there are readers remaining, the counter is greater than 0 - again we have two easily distinguishable states.

The synchronization variable is split in two parts: the writer active bit and the reader counter (Figure 4). We will exploit the properties of binary numbers to manipulate the counter - by adding 2 to the synchronization variable, the counter increases by one. When working with multiples of 2, the writer active bit will remain undisturbed. Atomic operations are readily available for adding, subtracting and testing bits.

Figure 4: Partition of the synchronization variable



The read lock procedure (Listing 12) increments the reader count even before checking the lock state. This is allowed, as long as we honor the writer active bit (defined by the macro `WAFLAG`), although it makes the algorithm even more biased towards readers. Now waiting readers always block writers. Note that the writer state is being checked with an ordinary read. As this is happening after the reader count is incremented[7], no new writers can jump in.

Writer lock (Listing 13) is identical to the spinlock implementation we gave before. Here the CAS operation is no longer optional - we're not allowed to destroy the reader counter, so a blind overwrite like we used in our assembly language versions does not work.

[7]There is the possibility that the compiler or the processor reorders ordinary reads or writes. Here this won't happen because the atomic add is considered to be a memory barrier.

Listing 12: Acquire a reader lock

```
#define RC_INCR 2
#define WAFLAG 1

/* Increment reader count atomically */
fetch_and_add(syn_var, RC_INCR);

/* Try getting the lock without pause */
if (!((*syn_var) & WAFLAG)) return;

/* Spin loop */
for (;;) {
  for(i=0; i<SPIN_COUNT; i++) {
    MM_PAUSE
    if (!((*syn_var) & WAFLAG)) return;
  }
  /* sleep for a bit (not shown) */
}
```

Because reader and writer unlocks are one-liners, we've grouped them together (Listing 14).

7 Lock-free FIFO

The final subject is the lock-free data structure. The idea is that the data should be accessed in such a way that it does not get corrupted, even if threads modify it concurrently. This results in some nice properties, for example one thread dying does not usually lock out all others. Lock-free synchronization does not necessarily outperform lock-based methods.

Locks, whether they are a critical section mutex, a reader-writer lock or something else, are a universal and generic way to synchronize access. On the other hand, lock-free synchronization is always specific to the task at hand, so each new problem needs a new synchronization algorithm.

Because the queue happens to be among the more reasonable structures to invent lock-free algorithms for, we will recycle our race condition problem from Section 2. We'll again cover the addition to the queue, as this is already complicated enough on its own (Listing 15). The algorithm was published in [5].

The technique is to loop until we manage to hook the new element to an existing queue element that does not yet have a successor. This kind of retry-until-CAS-hits-the-expected-value looping is rather typical of lock-free algorithms.

The goal to reach is on line 13. If we manage to catch an element that has NULL in the next pointer, we've successfully appended the element to the current end of the queue. This is so, because all other threads use the same

Listing 13: Acquire a writer lock

```
/* First attempt at getting the lock without spinning */
if (compare_and_swap(syn_var, 0, WAFLAG))
      return;

/* Spin loop */
for (;;) {
  for (i=0; i<SPIN_COUNT; i++) {
    MM_PAUSE
    if (!(*syn_var) && compare_and_swap(syn_var, 0, WAFLAG))
      return;
  }
  /* sleep for a bit (not shown) */
}
```

Listing 14: Reader and writer unlocking

```
/* reader unlock */
fetch_and_add(syn_var, -RC_INCR);

/* writer unlock */
atomic_and(syn_var, ~(WAFLAG));
```

function and the only way an element with no successor can end up in the queue is through the same CAS operation which atomically overwrites the previous NULL.

Getting to that line needs some preparation. First we will record the state of the two pointers we're preparing to overwrite at the moment we enter the loop. Line 8 makes sure that `next` belonged to the element pointed to by `tail` at the moment we read it. This is so because `t` equals `*tail` before and after the read.

Line 9 checks whether the pointer was NULL. If not, then because of line 8 we know that we caught some other thread between lines 13 and 16: the next element is already added, but `tail` is not yet updated. Again, this refers to the snapshot in time when we read those variables, but if the other thread happens to be pre-empted, the situation may very well persist. But, if the thread that is supposed to update the tail pointer is stalled then it would be useless to execute line 13 right now: our thread would just keep looping.

To address that, the algorithm includes a cooperative step where we attempt to advance the tail pointer by one element to "help" the stalled thread. If the CAS fails, some other thread already advanced the pointer, which is fine. In either case we will need to start over with the new tail pointer. This kind of behaviour might seem odd coming from the world of locks, where everyone just tries to grab as much as possible, but ultimately the helping thread benefits too so this isn't pure altruism.

Listing 15: Lock-free implementation of adding to the FIFO

```
1  void add_elem(struct elem **tail, struct elem *e) {
2    struct elem *t, *next;
3    for (;;) {
4      t = *tail;
5      __sync_synchronize();                    /* avoid "optimizing" access to tail */
6      next = t->next;
7      __sync_synchronize();                    /* this order must be preserved as well */
8      if (*tail != t) continue;
9      if (next) {
10       compare_and_swap(tail, t, next);
11       continue;
12     }
13     if (compare_and_swap(&(t->next), NULL, e))
14       break;
15   }
16   compare_and_swap(tail, t, e);
17 }
```

Once the new element is connected to the queue, the tail pointer can be updated. It is entirely possible that some other thread has already jumped in, read the pointer we've just written and copied it to `tail`. Again, the final CAS failing is perfectly fine, this just means it was no longer needed.

All this may be quite exciting. Unfortunately, this implementation only works if we *never* free the memory of the elements when they are removed from the queue: a rather prohibitive demand. The problem is that the CAS operations are vulnerable to the "ABA problem" - they do not guarantee that a value has not changed, they only guarantee that it is the same as before. For example, it is entirely possible that in the time span between executing lines 8 and 13, the tail element travels through the queue, is removed, freed and allocated again for some unrelated purpose that happens to write NULL at the offset we think the `next` pointer resides. Then we would in one stroke lose the element we're supposed to be adding and overwrite some unrelated data with garbage.

The memory management problem haunts various lock-free or otherwise non-blocking algorithms and complicates their deployment. One practical solution is to use "hazard pointers": an array global to all concurrent threads that contains pointers which should not be reused yet. Also, each thread has a local list of pointers that are waiting to be freed. Each time freeing an element is requested, it is appended to this list and then the entire list is checked against the array of hazard pointers. Those not classified as "hazardous" anymore will be recycled, for example with `free()`. Others will remain in the local waiting list. [5]

8 What to take home from here

The main purpose of presenting the program code and algorithms was educational. They reveal how parallel processes interact with each other and the hardware. As stated, the problems they solve have already been solved many times over in various programming languages, operating systems and libraries.

Yet for the adventurous it is also possible to put these examples into practical use. The spinlock implementation given has been used extensively in the main memory database WhiteDB [8] and should be rock solid, as far as we can tell. As always, only testing can tell if and how it fits into other applications.

The reader-writer lock has also been tested heavily, but suffers from performance problems under heavy load, most notably writer starvation. It is, however, very fast in situations when it is not continuously contented and where its main purpose is to avoid accidental corruption.

The lock-free queue implementation has not seen any real use; not to mention that it is only half of the solution. The algorithm it was based on is peer reviewed and well known, though. The paper by M.M. Michael [5] in the bibliography section contains the full algorithm and references to relevant publications.

Bibliography

[1] Pierre-Jacques Courtois, Frans Heymans, and David Lorge Parnas, *Concurrent control with readers and writers*, Communications of the ACM **14** (1971), no. 10, 667–668.

[2] JR Goodman and HHJ Hum, *Mesif: A two-hop cache coherency protocol for point-to-point interconnects*, Tech. report, University of Auckland, 2004.

[3] Intel Corporation, *Ap-949 using spin-loops on intel pentium 4 processor and intel xeon processor*, 2001.

[4] John M Mellor-Crummey and Michael L Scott, *Scalable reader-writer synchronization for shared-memory multiprocessors*, ACM SIGPLAN Notices, vol. 26, ACM, 1991, pp. 106–113.

[5] Maged M Michael, *Hazard pointers: Safe memory reclamation for lock-free objects*, Parallel and Distributed Systems, IEEE Transactions on **15** (2004), no. 6, 491–504.

[8] `http://whitedb.org`